Oscillation Control in Logic Simulation using Dynamic Dominance Graphs*

Peter Dahlgren
Department of Computer Engineering, Chalmers University of Technology
S-412 96 Gothenburg, Sweden

Abstract - Logic-level modeling of asynchronous circuits in the presence of races frequently gives rise to oscillation. A new method for solving oscillation occurring in feedback loops (FLs) is presented. First, a set of graph traversal algorithms is used to locate the FLs and order them with respect to a dominance relation. Next, a sequence of resimulations with the feedback vertices forced into stable states is performed. The proposed method can handle noncritical races occurring in asynchronous circuits and has applications in feedback bridging fault simulation.

I. INTRODUCTION

Oscillation control is the process of detecting oscillation during simulation and taking the appropriate corrective actions. There are many situations that cause uncontrolled oscillation in logic-level modeling (at the gate or switch level) of circuits that contain *Feedback Loops* (FLs). For instance, standard logic-level models lack the capability of modeling noncritical races in asynchronous circuits. Noncritical races are common in redundant networks, such as self-checking logic [1], in which there are several legal stable states.

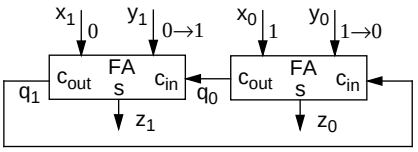
The oscillation control in existing simulators consists only of a detection mechanism, which is triggered when the number of events exceeds a predefined limit or when too long time has passed without reaching a stable state [2]-[4]. The oscillating signals are set to the unknown logic state (X), and no further analyzing is performed to solve the situation. Any circuit node logically sensitized by an X state node is also set to X. This spread of X values is a pessimistic approach and implies that logic-level design verification of many types of asynchronous circuits is not possible. Several studies have addressed the management of X values [5]-[8]. However, these methods are not sufficient when oscillating signals are present. No systematic strategy for handling oscillation in FLs has so far been proposed.

This paper presents a general method for global oscillation control in feedback loops consisting of several logic stages. Any number of oscillating FLs can be handled, and a graph that captures the logical dependencies among the FLs is dynamically created when oscillation occurs.

In the context of fault simulation, the bridging and transistor stuck-on fault models are known to be representative for a sizeable class of failures occurring in CMOS circuits [9]. These fault types often introduce FLs dynamically for certain input patterns [10]-[14]. If an FL consists of an even number of inverting stages, called an EFL, an originally combinational or synchronous circuit is then likely to become a circuit with asynchronous "memory" behavior.

In a single EFL, the logical effects can be traced to the primary outputs by forcing the EFL into the two possible stable internal states and resimulating the network twice. On the other hand, if the FL consists of an odd number of inverting stages, it is uncertain whether the loop will oscillate or settle in a stable state, possibly with some nodes set to intermediate values. Which of the two situations occurs depends on the electrical characteristics, such as the effective propagation delay of the loop relative to the rise and fall times [15], which are difficult to capture at the logic level in general situations.

For example, consider the end-around carry (eac) adder in Fig. 1a, which is commonly used for one's complement arithmetic. The input vector transition (0011) → (0110) in ($x_1y_1x_0y_0$) results in oscillation in a standard logic-level simulator. Given the initial state ($q_1\ q_0$)=(01), the right adder generates q_0^+=0 and the left adder generates q_1^+ = 1. In the next step, the two adders generate q_0^+ = 1 and q_1^+=0 and the network is back into its initial state. Hence, an oscillation between the two unstable states (01) and (10) will occur, as can be seen in the simplified flow table in Fig. 1b, in which there is a cycle in the column for the input vector (0110). However, any of the two stable states ($q_1\ q_0$)=(00) or (11) is an acceptable steady-state response because they result in the output patterns ($z_1\ z_0$)=(11) and (00), respectively, both of which represent the value zero. In a real circuit, the race that occurs for the transition (01)→(10)→(01) etc. in ($q_1\ q_0$) will result in the network reaching one of the stable states irrespective of the effective propagation delays of the two adders. However, the idealistic timing model at the logic level prevents the network from reaching a stable state. There is no model described in the literature that can handle this phenomenon. When several FLs are included in the network, the situation becomes even more complicated because the FLs may be logically dependent on one another.



(a)

q_1q_0	$x_1y_1x_0y_0$ 0000	0011	0110	0111	0010	...
00	[00]	01	[00]	01	[00]	
01	00	[01]	10	11	00	...
11	00	01	[11]	[11]	01	
10	00	01	01	01	01	

(b) Next state ($q_1^+\ q_0^+$) [] = stable state

Fig. 1. (a) Oscillation in an end-around-carry adder. (b) Flow table.

*) This work was funded by the Swedish Research Council for Engineering Science (TFR) under contract #95-608.

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

The method for oscillation control proposed in this paper consists of four operations: 1) detecting the oscillation, 2) locating the FLs, 3) analyzing dependencies among the FLs and 4) resimulating the network with feedback loops cut.

In the first operation, a directed graph called the Y graph is constructed. In this graph, all oscillating FLs and their logical implications on sensitized combinational paths are represented. During the next operation, an algorithm identifies the cycles in the Y graph, a process similar to the problem of finding a minimum set of feedback vertices in a directed graph [16]-[18]. If there exists more than one cycle, the third operation analyzes the logical and structural dependencies among the cycles and orders the cycles with respect to a dominance relation. The graph of this relation is then used to determine in which order the FLs must be processed.

For example, a common building block in error-detecting VLSI systems is the residue generator, which generates the check symbol to the residue code [1][19]. A residue generator is often implemented as a tree of eac adders, as shown in Fig. 2. In the generator in Fig. 2a, oscillation can occur in all three EFLs and the behavior of the lower EFL is logically dependent on the upper two EFLs, which means that the two *dominant* EFLs must be processed first.

Finally, in the fourth operation, the feedback loops are broken and a series of resimulations with the feedback nodes set to definite logic values is performed.

Section II reviews the event-driven algorithm used as the basis for the proposed simulation strategy. Section III presents a method for oscillation detection and defines the Y graph. Section IV presents a graph traversal algorithm for locating FLs in the Y graph. Section V concentrates on the logical dependencies among the FLs, and a graph-based strategy for systematically processing the FLs is proposed in Section VI. Finally, some simulation results are given in Section VII.

II. Basic Logic-Level Simulation Algorithm

The main procedure of a standard event-driven logic simulator is outlined in Fig. 3 (see for instance [2][20]). This procedure is performed for each input vector. Initially, all input nodes that change state are put in a time ordered queue (**ev_list**).

The **predict_node_event** procedure tests the new situation that has arisen as an effect of the node value assignment on a gate input node by computing the output value of that gate. An event is generated and put in **ev_list** if the new logic value differs from the current value of the node. The standard set of logic states is: L, H and X, where the L (low) and H (high) states are referred to as *definite logic states*. A gate in the basic algorithm may be a *complex gate* that implements an arbitrary logic function. A network is represented by a number of interconnected complex gates.

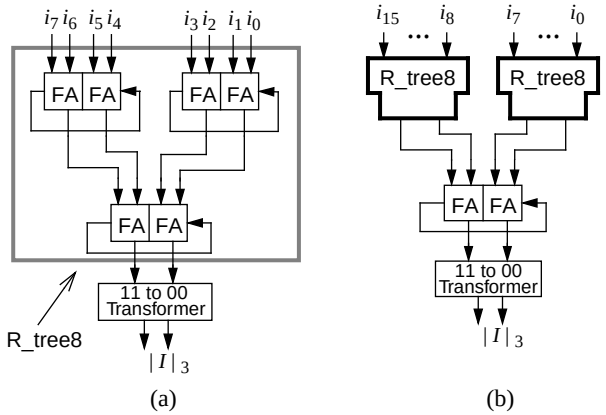


Fig. 2. Modular implementation of an (a) 8-bit and (b) 16-bit residue-3 generator.

```
simulate(ev_list)
    While (ev_list not empty)
        let nd =get_event(ev_list)
        set_node_state(nd)
        Foreach ( Fan out Gate, G, driven by nd)
            predict_node_event(G, nd)
        end foreach
    end while
```

Fig. 3. Basic event-driven simulation algorithm.

Furthermore, by introducing the concept of *Channel-connected Blocks*, CBs (or transistor groups), [5], in switch-level networks, it is possible to describe algorithms that are applicable to both the switch and gate levels because the signal propagation between two CBs is unidirectional. Only channel-connected MOS transistors are included in a CB, and any gate terminal of a transistor within a CB is defined as an input to that block. An event at the gate level corresponds to an *inter-CB propagation* at the switch level, which may consist of several local events within a CB. The algorithm descriptions that follow use complex gates (CGs) as the smallest components. It is therefore irrelevant whether a CG represents a logic gate or a CB. Details about the simulator into which the proposed algorithm was implemented are given in [21].

III. Detection and Representation of an Oscillating Circuit, the Y Graph

The detection of oscillation can be performed by counting the number of events generated. Oscillation has occurred when the event counter exceeds a predefined value, N_{CGmax}. In order to abort the oscillation and to further analyze the feedback situations that cause the oscillation, a graph called the *Y graph* is defined. By introducing an additional logic state, the Y state, this graph can be constructed automatically when the oscillation is detected by a few simple modifications of the basic algorithm. The Y state is similar to the X state when applied as input to a CG but can prohibit the event generation. The modifications of the basic algorithm in Fig. 3 are as follows:

(1) If the maximum number of events has exceeded N_{CGmax} then:
  (a) In **set_node_state**, if the logic state predicted for **nd** differs from the current state, set the node to the Y state.
  (b) In **predict_node_event**, if the node is already assigned the Y state, prohibit event generation on that node.

(2) If the number of events is less than N_{CGmax} then:
During the simulation, any Y state node is changed to the X state and the node evaluation continues in the normal mode.

By this scheme, the event generation will eventually fade out and a steady state will be reached. Any output node that is logically dependent on any of the oscillating nodes will be set to the Y state. When a new input vector is applied, the current FLs may be cut and/or new FLs may be created. Modification (2) together with setting the event counter to zero for each new input vector assure that the event generation will continue normally when a new input vector is applied.

All nodes assigned the Y state in the original network can be viewed as a directed graph where each vertex, v_i, corresponds to a CG, CG_i, whose output node is assigned the Y state. A directed edge exists from vertex v_i to v_j iff: i) the output of CG_i is directly connected to the input of CG_j in the corresponding network and ii) the output of CG_j is logically sensitized by the output of CG_i. (ii) means that the Y state input to CG_j must not be logically masked by the other inputs of CG_j. A directed cyclic path corresponds to a potentially oscillating FL in the original network. The Y graph forms a dynamic model of the oscillation situation for the current input state and includes the

logical implications of the FLs on all sensitized signal paths in the original network.

Fig. 4a shows the Y graph of the oscillating adder in Fig. 1. A full adder implementation that consists of four CGs (the inverse carry and sum generation blocks and two inverters [6]) was used. The definition of the Y graph implies that the primary inputs are never included in a Y graph. Note that the two edges e_1 and e_2 in Fig. 4a correspond to the same network node (q_1), which is connected to the input of two subsequent CGs. Fig. 4b shows a simplified Y graph in which each full adder has been replaced by a single CG.

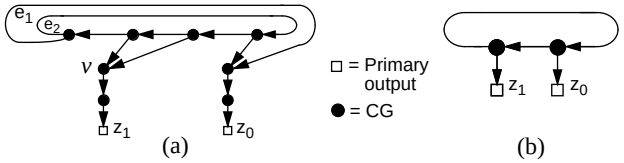


Fig. 4. (a) Y graph for the oscillating eac adder. (b) Simplified Y graph.

IV. Locating Feedback Loops in a Logic Network

Locating the FLs that are potentially oscillating corresponds to finding all directed cycles in the Y graph. Only one cycle through any vertex is allowed. This single-cycle restriction is imposed because it must be possible to order the cycles with respect to a dominance relation prior to the processing of them (see Section V). Finding cycles in a Y graph is similar to the problem of finding a minimum set of feedback vertices, which is known to be an NP complete problem [22]. However, the single-cycle restriction significantly reduces the ways in which cycles can be formed. Moreover, a set of feedback vertices is not sufficient to represent the cycles; all vertices belonging to each cycle must be considered (see Section V).

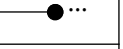
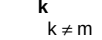
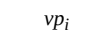
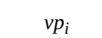
Here, a graph traversal algorithm is presented that repeatedly tries to build cycles by assigning a unique number to all vertices belonging to a particular path. A predecessor v_p of a vertex v_s is a vertex for which there exists a directed edge from v_p to v_s. The graph tracing is performed in reverse order of the directed edges, starting with the output vertices and recursively continuing with the predecessors. This is basically a depth-first search with a termination condition based on a vertex enumeration scheme. Each vertex can be assigned three different types of identifiers: 1) no assignment, 2) a number, m, or 3) a number with a bar, $\overline{m}$. A global variable *gc* is used to select new numbers for the vertices when there is more than one predecessor. The variable *gc* can be increased by one at any instance of the algorithm. The **trace** algorithm, given in Fig. 5, is invoked for one output vertex at a time. Before each of these initiating calls is performed, *gc* is increased by one and the output vertex is set to the current value of *gc*.

trace(*v*: vertex)
  Foreach (*predecessor, vp_i , of v*)
    check vertex vp_i and perform operations according to Table 1 (Conditions 1-4*).*
  end foreach
mark_cycle(*v*: vertex)
  Foreach (*predecessor, vp_i , of v*)
    check vertex vp_i and perform operations according to Table 1 (Conditions 5-7*).*
  end foreach

Fig. 5. Algorithm: Locating feedback loops.

Condition 3 in Table 1 indicates that a cycle has been found for the path numbered m and a new procedure, **mark_cycle**, is initiated which searches through the cycle once more and marks all vertices that belong to that cycle by $\overline{m}$. In Conditions 1 and 2 in Table 1, *gc* is increased by one and a new path is started if there is more than one predecessor in the Y graph. For instance, vertex *v* in Fig. 4a has two predecessors. Condition 4 indicates that a vertex that already belongs to a cycle has been reached and the single-cycle restriction implies that the search along this path can be terminated. Condition 6 means that the entire cycle has been located in the **mark_cycle** procedure and all vertices have been marked $\overline{m}$.

Table 1. Primitive operations performed for various situations in the **trace** (Conditions 1-4) and **mark_cycle** (Cond. 5-7) algorithms.

Cond.	Current situation	Operations performed
1		*If vp_i is first predecessor at this instance of* **trace**: a)  b) **trace**(vp_i)
2		*else*: a) gc=gc+1 b)  c) **trace**(vp_i)
3		a)  b) **mark_cycle**(vp_i)
4		*No operation*
5		a)  b) **mark_cycle**(vp_i)
6		*Save vertex vp_i as FV for cycle $\overline{m}$.* *Log the cycle length.* *Exit from* **mark_cycle.**
7	*All other situations*	*No operation*

As an example, Fig. 6 shows the **trace** algorithm applied to the Y graph of the residue-3 generator in Fig. 2a for the situation in which all three FLs are oscillating. The boldfaced numbers indicate the current value of the vertices and the encircled numbers show the order in which the various steps are carried out in the **trace** algorithm. Steps 1-4 in Fig. 6a all satisfy Condition 1 in Table 1. In step 5, cycle C_1 is found and marked, i.e. Condition 3 is satisfied and vertex *a* is logged as the *Feedback Vertex* (FV) of C_1. Next, in Fig. 6b, vertex *b* is subjected to Condition 4 and not processed. As step 6, there is therefore only one predecessor to select which is assigned the value 2. There are three possible predecessors to select, as step 7. Assuming that the path to vertex *c* is selected, cycle C_2 is found in step 9 and *c* is logged as an FV. In step 10 in Fig. 6c, Condition 2 is satisfied and the vertex number at *d* is overwritten. Finally, in step 12, cycle C_3 is found with *d* logged as an FV. No further paths are possible to trace because all predecessors of any vertex are subjected to Condition 4. Next, calling trace from output vertex I_0 results in vertex *e* being set to 4. No more paths are possible to trace, and the final state is given in Fig. 6d.

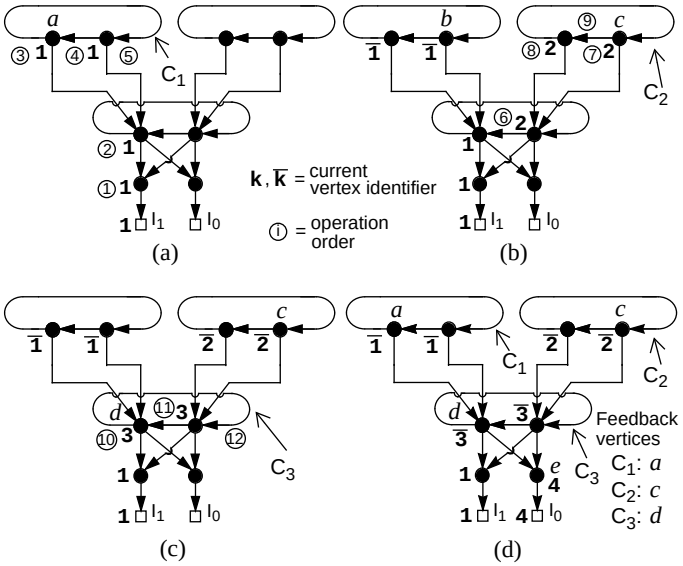


Fig. 6. (a)-(c) The **trace** and **mark_cycle** algorithms applied to the residue-3 generator. (d) Final state.

Fig. 7a shows an example of another asynchronous circuit. Let the output z_0 of the eac adder in Fig. 1a be connected to the input as shown in Fig. 7a. Moreover, assume that the adder is oscillating and that c=0 (meaning that s=r=1). Fig. 7b shows the resultant Y graph when the transition 0→1 occurs at c. Applying the **trace** algorithm results in either the situation in Fig. 7c or 7d, depending on the order in which the predecessors are processed. The fact that the set of cycles and FVs is not unique is of no major importance as cycles that are mutually dependent on one another are merged into a single cycle, as described in Section V.

V. Ordering of Cycles under a Dominance Relation

Given that the cycles in the Y graph have been located, the next step is to process the FLs by forcing the network nodes corresponding to FVs into definite logic values and resimulating the network. However, when there are several cycles, the processing of the FLs must be performed in a certain order because there may be logical dependencies among various FLs. Furthermore, several FVs may be processed simultaneously if they are independent of one another. A dominance relation is therefore defined that describes the logical dependencies among the cycles in the Y graph.

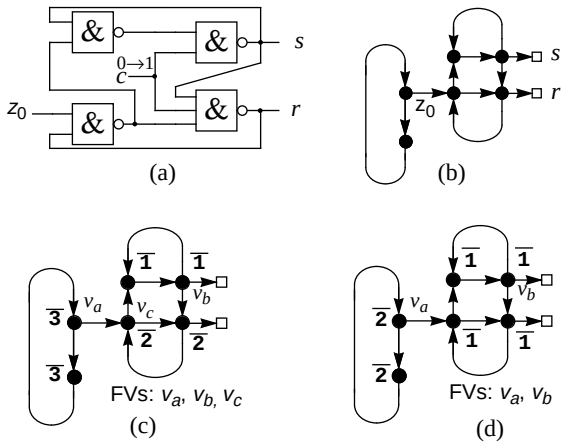


Fig. 7. (a) An asynchronous circuit that results in overlapped cycles. (b) Y graph. (c)-(d) Results of **trace** algorithm.

A. Dominance between Feedback Loops

A cycle C_1 *dominates* another cycle C_2, written as $C_1 \rightarrow C_2$, if there exists a directed path in the Y graph from a vertex in C_1 to a vertex that belongs to C_2. For example, the dominance relations in Fig. 6d are: $C_1 \rightarrow C_3$ and $C_2 \rightarrow C_3$. The dominance relations between all connected cycles in a Y graph can be obtained by the traversal algorithm given in Fig. 8. For a given cycle, all paths originating from vertices belonging to the cycle are traversed in reverse order depth-first, and the search terminates when either another cycle is found or the path returns to the original cycle.

```
cycle_depend
    Foreach (cycle, C)
        let ū denote the value of the vertices in C
        Foreach (vertex, vc_i , that belongs to C)
            search_dominant_cycles(vc_i , ū)
        end foreach
    end foreach
search_dominant_cycles(v: vertex, z̄: vertex_number)
    Foreach (predecessor, vp_i , of v)
        if (vertex identifier of vp_i is of type k̄)
            then
                if ( z̄ ≠ k̄ ) then    Save relation "C_k̄ → C "
                                      (* cycle k̄ dominates C *)
                end if
            else    search_dominant_cycles(vp_i , z̄)
        end if
    end foreach
```

Fig. 8. Algorithm: determining cycle dominance relations.

B. Cycle Dominance Graphs

The set of relations between neighboring cycles obtained by the **cycle_depend** algorithm can be represented as a *Cycle Dominance Graph* (CDG). In a CDG, each vertex represents a cycle in the Y graph, and there is a directed edge from vertex C_k to C_p iff $C_k \rightarrow C_p$.

The assignment of definite logic values to the nodes in one loop that dominates another loop can result in the elimination of the dominated FL owing to logical properties of the Y state (which are identical to those of the X state). Given that there are no directed cycles in the CDG, there is a unique order in which the FVs must be processed without violating the logical implications. A root vertex of a CDG is a cycle in the Y graph that is not dominated by any other cycle. Fig. 9a shows the CDG of the Y graph in Fig. 6d, and Fig. 9b-c shows the CDGs for the two sets of cycles in Fig. 7c-d.

Given that the CDG is acyclic, let the *vertex distance* $[C_p]$ of a vertex C_p be defined by the *maximum* number of vertices that can be included in a directed path between a root vertex and C_p. All FLs that correspond to vertices which are located at the same distance in a

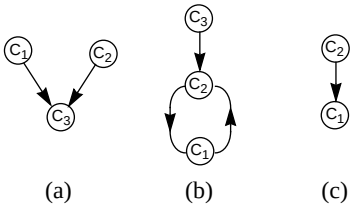


Fig. 9. Examples of cycle dominance graphs.

CDG can be processed simultaneously because they are logically independent of one another. The distance to each vertex can be obtained by the **get_distance** algorithm given in Fig. 10, which searches through all directed paths depth-first and assigns a number to each vertex. The number is increased by one each time a vertex is passed and, if the number is less than or equal to the current value of the vertex, the search is terminated. Initially, assume that all vertices have been assigned the value zero. Next, **get_distance** is called from each root vertex with a zero as argument.

As an example, Fig. 11 shows the **get_distance** algorithm applied to a CDG. Fig. 11a shows the values at the vertices after the call **get_distance**(C_a,0), and Fig. 11b shows the final state after the call **get_distance**(C_j,0).

All vertices in a CDG can be divided into classes with respect to the vertex distance. Let the *vertex distance class* D_k be defined by all vertices C_p for which $[C_p]=k$. Fig. 11c shows the vertex distance classes of the example.

If there are directed cycles in a CDG, they must be eliminated before the vertex distance classes can be determined because directed cycles indicate that there are mutual dependencies among various FLs. All vertices belonging to a directed cycle in the CDG are replaced by a single vertex. This is the reason for the single-cycle restriction in the **trace** algorithm. The process of locating cycles in a CDG is similar to the **trace** algorithm in Section IV. For instance, in Fig. 9b, the two vertices $C_1$and C_2 are merged into a single vertex and the topology of the resultant graph becomes identical to that shown in Fig. 9c.

```
get_distance(C: vertex, n: number)
    Let n = n+1
    Foreach (successor, C_s , of C)
        if ( n > current value of C_s )
            then    Assign the value n to C_s
                    get_distance( C_s , n )
        end if
    end foreach
```

Fig. 10. Algorithm: Computing vertex distance classes.

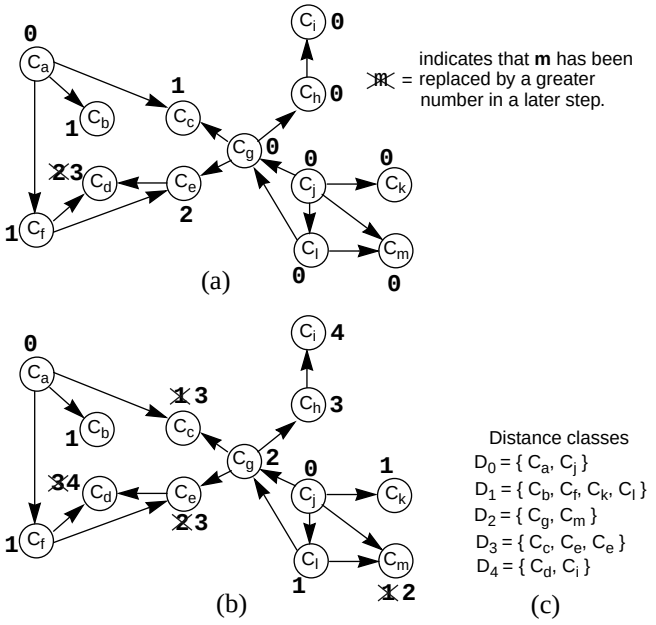


Fig. 11. (a)-(b) Determining the vertex distances in a CDG. (c) Resultant distance classes.

VI. Processing Feedback Loops and Resimulation

Two approaches are proposed for the solution of FLs represented as a CDG: *A*) finding one stable solution and *B*) evaluating all possible solutions.

A. Finding one Stable State

The basic principle here is to cut all FLs corresponding to the vertices in a given distance class, to force the FVs associated with each FL into a definite logic state and then to resimulate the network. The distance classes, D_i, are processed one at a time in increasing order of the class index *i*. As only one solution is considered, the logic state to which an FV is set can be chosen as the latest definite logic state assigned to the corresponding network node. The maximum number of simulations is given by the number of vertices in the longest directed path between any root vertex and a leaf vertex in the CDG. In the procedure given in Fig. 12, let *L* denote a list of all FVs and the corresponding definite logic states to which they will be set. L_k denotes the list of logic state assignments for the vertices that belong to the distance class D_k.

```
process_loops(D: set of distance classes, L: list of node states)
    let i = 0
    let max_dist = number of distance classes
    while (network is not free from oscillation and i ≤ max_dist )
        cut_and_gen( D_i , L_i)
        simulate( ev_list )              (* See Fig. 3 *)
        i = i + 1
    end while
cut_and_gen( D_k: vertex distance class, L_k: list of node states )
    Foreach ( vertex, C_i , in D_k )
        let nd = the network node corresponding to the FV of C_i
        if ( nd is not assigned a definite logic state)
            then    set nd to the logic state L_ki
                    put an event for nd in global ev_list
        end if
    end foreach
```

Fig. 12. Algorithm: Processing feedback loops.

The procedure **process_loops** is aborted as soon as a stable solution has been found, regardless of the number of distance classes processed. Furthermore, loops corresponding to vertices at a greater distance class number may be stabilized owing to the cycle dominance. This is checked in **cut_and_gen**, in which the logic state is changed and an event is generated only for those network nodes whose values are still oscillating. For example, in the situation shown in Fig. 9a, the FVs corresponding to C_1 and C_2 are processed simultaneously in the first step. In the next step, when C_3 is to be processed, this FL may already be stabilized, meaning that no further operations are necessary.

B. Evaluating all Possible Solutions

In the general case that all stable solutions must be found, the **process_loops** procedure must be called for each possible assignment of logic states to the vertex nodes in *L*. Next, the set of output vectors obtained for each solution of the algorithm in Fig. 12 must be evaluated. If there are output vectors for which the logic state of a given bit position differs, the value of this bit position must be set to X, which means that this output value is dependent on the internal state of the network. Of course, this approach, which is similar to evaluating all possible function values of an element with X values as input, may be very time-consuming if there are many cycles. However, in many applications, one stable solution is sufficient or only a few cycles need to be considered. For instance, in conventional bridging fault simulation, the Y graph consists of only a single cycle, which means that only two solutions are required.

VII. Experimental Results

The simulation strategy proposed in Sections III-VI was implemented in the switch-level simulator described in [21] and various asynchronous circuits were simulated. Tables 2 and 3 show the results of an experiment in which 1,000 randomly chosen input vectors were applied to the tree-based networks in Fig. 2. It can be seen that, for the traditional models, the oscillation resulted in output X values for 25%-70% of all input vectors applied. On the other hand, the proposed method predicted the correct results for all input vectors. Furthermore, it can be seen in Table 3 that, on average, two oscillating feedback loops were found for the 545 input vectors that gave rise to oscillation in the res_gen16 network. The strategy for finding one stable solution only (Section VI.A) was used. The low number of additional calls to **simulate** (639) implies that, in most cases, the network became stable after only one resimulation step in the algorithm in Fig. 12. The author is not aware of any other logic-level simulator that can handle the types of networks given in Fig. 2.

Table 2. Simulation results for traditional models.

Network	No. of output vectors containing X values	
	COSMOS [4]	Basic algorithm in Section II ([21])
res_gen8	291	266
res_gen16	686	615

Table 3. Simulation results for the proposed method.

Network	Oscillation detected		Total no. of calls to **simulate**[a]	Overhead, additional no. of events
	No. of input vectors	Average no. of FLs found		
res_gen8	227	1.26	1,231	5%
res_gen16	545	1.93	1,639	11%

a. In the basic algorithm, one call to simulate is done for each input vector.

VIII. Discussions

All operations but the assignment of definite logic states to FVs in Fig. 12 are also applicable to FLs containing an odd number of inverting stages. If more accurate timing information were provided, the proposed method could be extended to predict whether such FLs would oscillate or settle and, in the latter case, determine the stable state. Odd numbered FLs can then be managed in a way similar to that for EFLs.

IX. Concluding Remarks

A new method for handling oscillation occurring in feedback loops was presented. The proposed simulation strategy extends the class of asynchronous circuits that it is possible to analyze at the logic level by modeling races and has applications in the simulation of feedback faults. Two types of graphs were defined to represent all potentially oscillating loops and the logical dependencies among them. A set of simple graph traversal algorithms was presented to locate the feedback loops and to determine the order in which they must be processed. An arbitrary number of oscillating loops can be handled, and the method can easily be implemented in standard event-driven simulators.

Acknowledgments

The author would like to thank Eskil Johnson at Chalmers University of Technology and Peter Liden at AB Volvo for their valuable criticism and stimulating discussions.

References

[1] M. J. Ashjaee and S. M. Reddy, "On totally self-checking checkers for separable codes", *IEEE Trans. on Comp.*, Vol. 26, Aug. 1977, pp. 737-744.

[2] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Testing and Testable Design*, Computer Science Press, 1990.

[3] P. Agrawal and W. J. Dally, "A hardware logic simulation system", *IEEE Trans. on CAD*, Vol. 9, No. 1, Jan. 1990, pp. 19-29.

[4] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits", *Proc 24th ACM/IEEE Design Automation Conf.*, 1987, pp. 9-16.

[5] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE Trans. on Comp.*, Feb 1984, Vol. C-33, No. 2, pp. 160-177.

[6] P. Dahlgren, and P. Liden, "Efficient modeling of switch-level networks containing undetermined node states", *Proc. ACM/ IEEE Int. Conference on CAD*, 1993, (ICCAD-93), pp. 746-752.

[7] L. P. Huang and R. E. Bryant, "Intractability in linear switch-level simulation", *IEEE Trans. on CAD*, 1993, Vol. 12, No. 6, pp. 829-836.

[8] J. Gecsei and E. Cerny, "Self-adjusting networks for VLSI simulation", *IEEE Trans. Comp.*, Sept. 1987, Vol. C-36, No. 9, pp. 1114-1120.

[9] F. Ferguson and J. Shen, "Extraction and simulation of realistic CMOS faults using inductive fault analysis", in *Proc. IEEE Int. Test Conference*, Sept. 1988, pp. 475-484.

[10] S. Xu and S. Y. H. Sy, "Detecting I/O and internal feedback bridging faults", *IEEE Trans. Comp.*, June 1985, Vol. C-34, No. 6, pp. 553-557.

[11] P. C. Maxwell and R. C. Aitken, "Biased voting: A method for simulating CMOS bridging faults in the presence of variable gate logic thresholds", in *Proc. IEEE Int. Test Conference*, Oct. 1993, pp. 63-72.

[12] J. M Acken and S. D. Millman, "Accurate modeling and simulation of bridging faults", in *Proc. IEEE Custom integrated Conf.*, 1991, pp. 17.4.1-17.4.4.

[13] T. M. Storey and W. Maly, "CMOS bridging fault detection", *Proc. IEEE Int. Test Conference*, 1991, pp. 1123-1131.

[14] J. Rearick and J. H. Patel, "Fast and accurate CMOS bridging faults simulation", in *Proc. IEEE Int. Test Conference*, Oct. 1993, pp. 54-62.

[15] Y. K. Malai, A. P. Jayasumana and R. Rajsuman, "A detailed examination of bridging faults", in *Proc. IEEE International Conference on Computer Design*, 1986, pp. 78-81.

[16] G. W. Smith and R. B. Walford, "The identification of a minimal feedback vertex set of a directed graph", *IEEE Trans. on Circuit and Systems*, Jan 1975, Vol. CAS-22, No. 1, pp. 9-15.

[17] P. Ashar and S. Malik, "Implicit computation of minimum-cost feedback-vertex sets for partial scan and other applications", *Proc. 31st ACM/IEEE Design Autom. Conf.*, 1994, pp. 77-80.

[18] S. Chakradhar, A. Balakrishnan and V. D. Agrawal, "An exact algorithm for selecting partial scan flip-flops", *Proc. 31st ACM/ IEEE Design Automation Conference*, June 1994, pp. 81-86.

[19] D. Nikolos, A. M. Paschalis and G. Philokyprou, "Efficient design of totally self-checking checkers for all low-cost arithmetic codes", *IEEE Trans. on Comp.*, Vol. 37, No. 7, July 1988, pp. 807-814.

[20] M. A. d'Abreu, "Gate-level simulation", *IEEE Design and Test of Computers*, Dec. 1985, pp. 63-71.

[21] P. Dahlgren, "A multiple-dominance switch-level model for simulation of short faults", *Proc. ICCAD-95*, San Jose, CA, 1995, pp. 674-680.

[22] R. M. Karp, "Reducibility between combinatorial problems", in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds., New York: Plenum Press (1972), pp. 85-103.